CpE 468–Computer Architecture

Project Architecture

By:

Asmaa Alazemi

Ghanima Almufarrej

Radiyah Alkofaidi

Supervised by:

Dr. Imtiaz Ahmad

Eng. Aishah Alnoori

Department of Computer Engineering
College of Engineering and Petroleum
Kuwait University
2024 – 2025

10/May/2025

Table of Contents

Tables

Tables of Figures

Credits

Member	**Task**
Asmaa Alazmi	CPU algorithm, GPU optimized algorithm, introduction, methodology, testing of code and images , editing the codes, Python codes
Ghanima Almufarrej	GPU algorithm implantation, parallel GPU version, output GPU results, Conclusion
Radiyah Alkofaidi	Histograms, speed up

1. Introduction & Objectives

This project focuses on implementing the Local Binary Pattern (LBP) algorithm for texture analysis in images using CUDA C. LBP is a widely used texture descriptor in computer vision due to its simplicity, flexibility and usefulness in capturing the local patterns. The main objective of this project is to explore the capabilities of parallel programming in CUDA C and to use it to improve the computational performance of the GPU. A comparison will take place between the CPU (traditional approach) and the GPU (parallel approach). Different images with different sizes will be tested. Finally, a histogram will showcase all the results captured.

The key learning objectives of this project are:

- To understand and implement the LBP algorithm for texture feature extraction.
- To gain hands-on experience with CUDA C and GPU programming.
- To analyze and compare the performance between CPU and GPU implementations.
- To construct and interpret histograms of LBP values for texture classification tasks.

The algorithms were tested on the following system:

- CPU: Processor AMD Ryzen 7 7700 8-Core Processor, 3801 Mhz, 8 Core(s)
- GPU: GeForce RTX 3080
- RAM: 32GB

2. Methodology

The implementation will begin with developing the traditional CPU-based Local Binary Pattern (LBP) algorithm. All versions will process a grayscale image in the P5 Netpbm format, assuming the input is already in greyscale. The algorithm will compute the LBP value for each pixel by comparing it with its neighboring pixels, generating the LBP image. Execution time will be recorded to compare it with the other algorithms. Visual Studio Code 2019 will be used for the implementation.

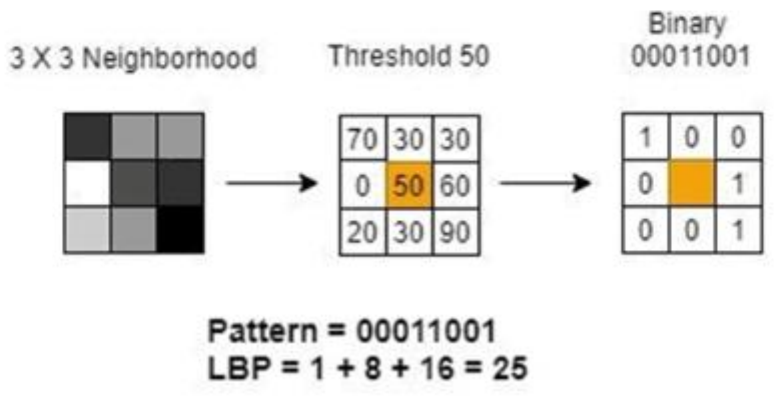

Figure 1 - How LBP works

2.1 Non-Parallel CPU Version

In the CPU implementation of the Local Binary Pattern (LBP) algorithm, the function iterates over each pixel in the image, skipping boundary pixels since they lack a full 3x3 neighborhood. For each valid pixel, it compares the intensity of the surrounding 8 neighbors to the center pixel. If a neighbor's value is greater than or equal to the center, a corresponding bit is set in an 8-bit binary pattern using weighted additions (e.g., top-left = 1, top-center = 2, ..., left = 128). These binary weights are summed to form a decimal LBP value, which is stored in the output image array.

2.2 Parallel GPU Version

Since the project was divided between us, after the CPU was implemented, it was converted to parallel, but not all steps are the same. The main difference between them is that the kernel used a parallel approach in indexing to compare the center pixel with its neighbor's intensity.

The parameters used for the Kernal call are the picture input, picture output and the width and height. First, a boundary check is used to make sure no index is out of bound when computing. The center is then saved in a variable to make comparing easier moving on with the next steps. Two loops were used to go through the 3x3 window. Then check again if pixels are out of bound, since before doing this step errors occurred. Each pixel in that neighborhood is then compared to the center. If it was bigger than or equal then LBP will be one, if not then zero. A check to make sure the center pixel does not compare its value to itself is implemented in the beginning of the loop, then lastly increasing the index of LBP after a comparison is made. The LBP is saved in an array to save the binary number as zeros and ones. Then converted to binary and stored in the output picture array outside of the loops. Errors did accrue and AL tools helped to detect them.

2.3 Optimized GPU Version

For this approach the image is divided into fixed size blocks 32x32, and each block is processed independently by a group of threads. To enable fast memory access and reduce the total execution time a larger region is loaded into the shared memory representing 'Tiles'. This region size is TILE_WIDTHxTILE_WIDTH which is 34x34. Why 34x34? Because the surrounding border pixels must be included.
Each thread is responsible for calculating the LBP value of a pixel within the block by comparing it to its neighbors in the shared memory tile. Additionally, we use a shared histogram within each block to accumulate LBP values, reducing global memory conflicts. Atomic operations are used in computing the histogram to avoid race conditions. This design ultimately increases the speed up in the LBP computation. [5, 6, 7]

2.4 Histogram

To capture the histogram graph of the LBP image for each approach, a 0 to 255 bin will be used as the x-axis (why 255? Because the maximum value of a byte is 255). And for the y-axis the number of pixels will be used. For the CPU implementation, this will involve iterating over each pixel in the LBP image and incrementing the count in the corresponding histogram bin based on the pixel's LBP value. For the GPU implementation, atomic operations will be used to safely update the histogram in global memory, allowing multiple threads to contribute without race conditions. In the optimized GPU version, shared memory will be used within each block to build partial histograms locally, which are then combined into the global histogram using atomic operations. So basically, to improve the code the kernel will handle the histogram in the GPU approaches. Finally, the resulting histogram will represent the frequency distribution of texture patterns in the image and will be used for the analysis and comparison part. The histogram will be generated using the following Python code:

```python
import matplotlib.pyplot as plt
import numpy as np

# Histogram values for shared memory
# add the values of the histogram for each image in the empty list []
histo_128 = np.array([])
histo_512 = np.array([])
histo_1024 = np.array([])

bins = np.arange(256)
# the graph y-axis is set to the maximum value of the histogram
max_pixels = max(histo_128)

plt.figure(figsize=(12, 6))

plt.bar(bins, histo_128, label='128x128', alpha=0.7)
plt.xlabel('Bin (LBP Value)')
plt.ylabel('Number of Pixels')
plt.title('LBP Histogram for 128x128 Image (GPU)')
plt.xlim(0, 255)
plt.ylim(0, max_pixels)
plt.legend()
plt.grid(axis='y', alpha=0.5)
plt.tight_layout()
plt.show()
```

3. Testing

All three implementations will be tested on images of sizes 128×128, 512×512, and 1024×1024, with comparisons made in terms of execution time, correctness of output, and the speedup achieved over the CPU and basic GPU versions compared to the improved GPU version. Regarding the image format the image will be PGM (P5), initially a greyscale image.

Figure 2 - Original greyscale image

The following Python code is used to convert the greyscale image to P5 (PGM) format:

```
import numpy as np
import PIL as Image

def convert_to_p5(input_image_path, output_pgm_path, size):
    # Open image in the correct format
    img = Image.open(input_image_path).convert("L")
    img = img.resize((size, size))

    # Convert the image to a numpy array
    image_data = np.array(image, dtype=np.uint8)

    # Save the image as P5 PGM format
    with open(output_pgm_path,"wb") as f:
        # Write the P5 header
        f.write(b"P5\n")
        f.write(f"{size} {size}\n".encode())  # Set the size to 400x400
        f.write(b"255\n")  # Max pixel value for PGM is 255
        f.write(image_data.tobytes())
```

3.1 CPU Algorithm Results

3.1.1 CUDA code

```
void cpuLBP(unsigned char* Arr, unsigned char* ArrOut, int height, int width) {
// CPU implementation for the LBP kernel ;D
// we will skip the boundary pixels because they dont have 8 neighbors, start with i=1
    for (int i = 1; i < height - 1; i++) //iterate through the rows
    {
        for (int j = 1; j < width - 1; j++) //iterate through the columns
        {
            unsigned char center_pixel = Arr[i * width + j]; // 1D access for 2D array
                                int lbp = 0; // initialize LBP value

            // check the neighbors

            /*
            B0  B1  B2
            B7  C   B3
            B6  B5  B4
            */

            if (Arr[(i - 1) * width + (j - 1)] >= center_pixel)
                lbp += 1; //B0
            if (Arr[(i - 1) * width + j] >= center_pixel)
                lbp += 2; //B1
                                if (Arr[(i - 1) * width + (j + 1)] >= center_pixel)
                                        lbp += 4; //B2
                                if (Arr[i * width + (j + 1)] >= center_pixel)
                                        lbp += 8; //B3
                                if (Arr[(i + 1) * width + (j + 1)] >= center_pixel)
                                        lbp += 16; //B4
                                if (Arr[(i + 1) * width + j] >= center_pixel)
                                        lbp += 32; //B5
                                if (Arr[(i + 1) * width + (j - 1)] >= center_pixel)
                                        lbp += 64; //B6
                                if (Arr[i * width + (j - 1)] >= center_pixel)
                                        lbp += 128; //B7

                                // convert the LBP value to binary (8-bits) byte
            unsigned char binary_lbp = static_cast<unsigned char>(lbp);
                        // store the decimal LBP value in the output array
                                ArrOut[i * width + j] = lbp;
                                histogram[binary_lbp]++; // update histogram
        }
    }

            }
```

3.1.2 Output images & Execution time

The results are for the size from 128 to 1024 (left to right):

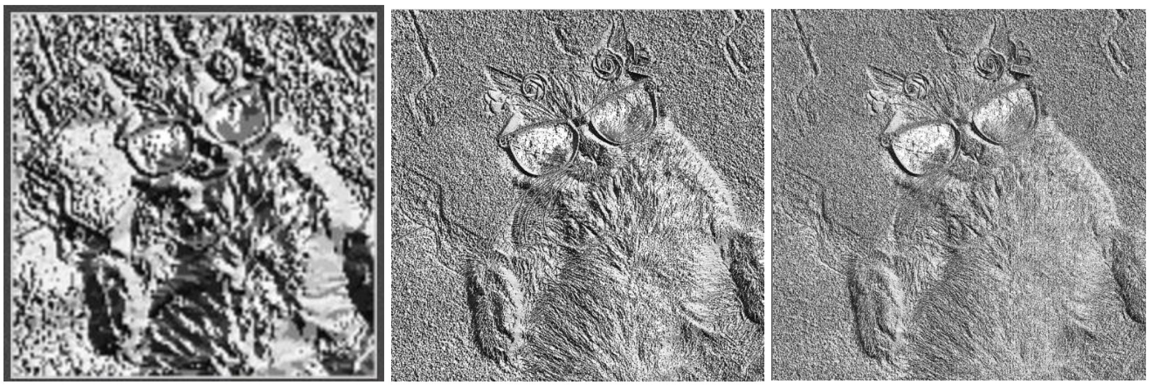

Figure 3 - LBP **CPU** output images

Table 1 - Execution time of the different inputs (CPU)

IMG SIZE	EXECUTION TIME (ms)
128X128	~0
512X512	6
1024X1024	22

3.1.3 Histogram

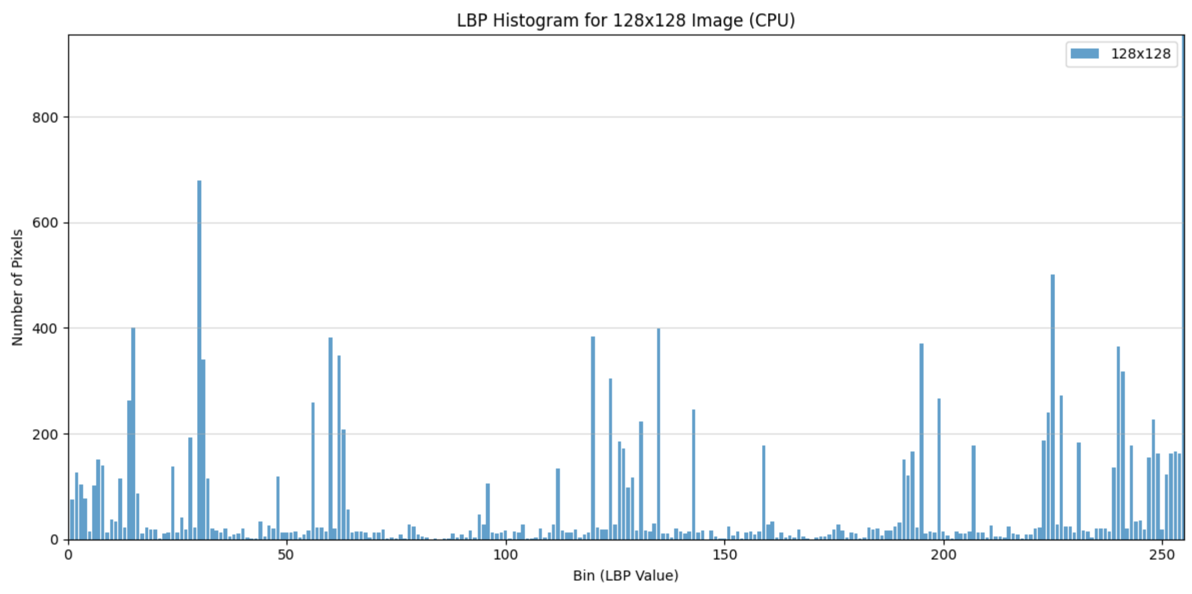

Figure 4 - CPU histogram 128x128

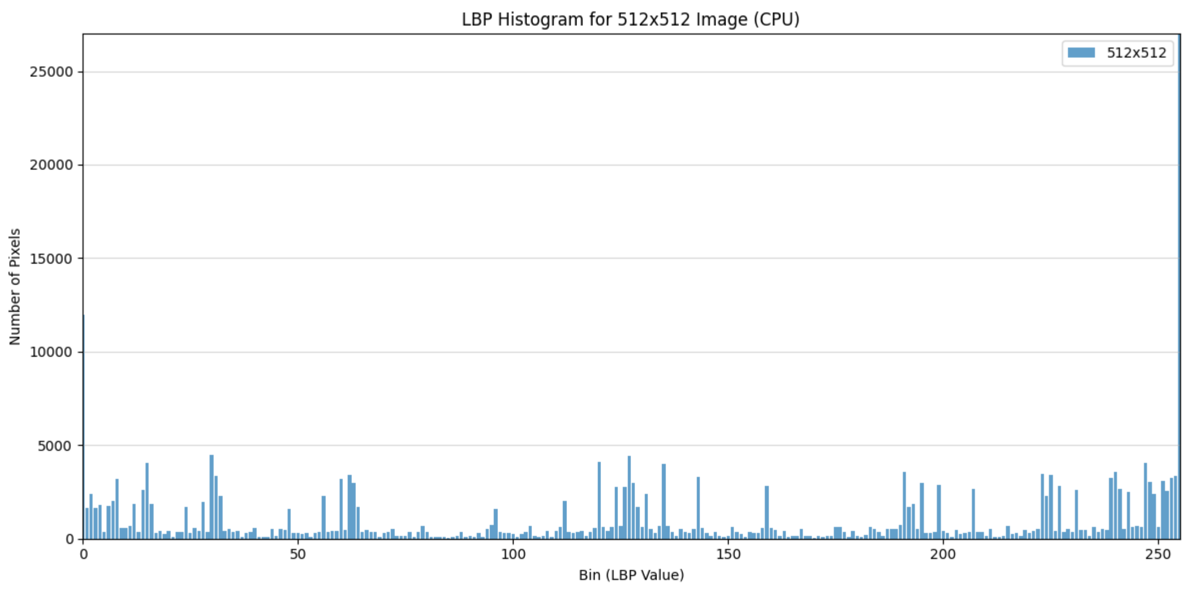

Figure 5 - CPU histogram 512x512

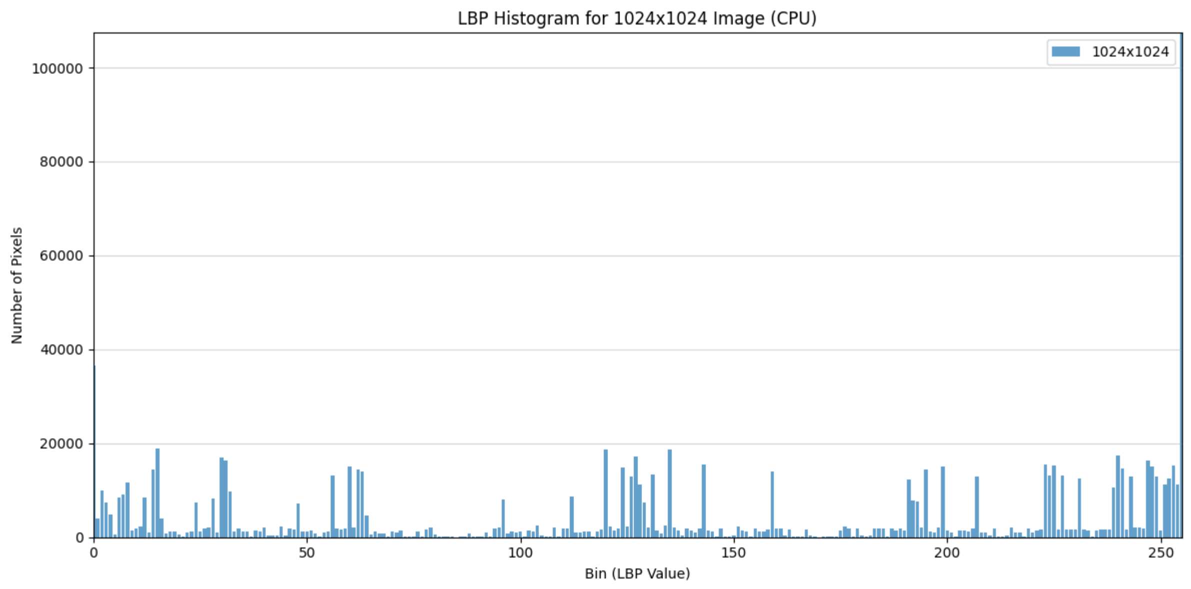

Figure 6 - CPU histogram 1024x1024

3.2 GPU Algorithm Results

3.2.1 CUDA kernel code

```
#define WIDTH 32 // same as num of thread

__global__ void greyscaleToLbpConversion(unsigned char* in, unsigned char* out, int w, int h, int* histo)

{

// Calculate global row and column indices based on the block and thread indices
        int bx = blockIdx.x;
        int by = blockIdx.y;
        int tx = threadIdx.x;
        int ty = threadIdx.y;
        int row = by * WIDTH + ty;
        int col = bx * WIDTH + tx;

        // decleration for LBP vars
        int LBP[8];
        int LBPValue = 0;
        int lbp_i = 0;

        if (row < h && col < w ) {
        unsigned char center_pixel = in[row * w + col]; // 1D access for 2D array

        for (int i = -1; i <= 1; i++) {
                for (int j = -1; j <= 1; j++) {
                        if (row + i >= 0 && row + i < h && col + j >= 0 && col + j < w) {
                                // check if nighbor pixel is center
                                if ((i == 0 && j == 0))
                                        continue;

                                // check neighbors with center
```

```
                    if (in[(i + row) * w + (j + col)] >= center_pixel)
                        LBP[lbp_i] = 1; // neighbor > center -> LBP = 1
                    else if (in[(i + row) * w + (j + col)] <= center_pixel)
                        LBP[lbp_i] = 0; // neighbor < center -> LBP = 0

                        // increment LBP index
                        lbp_i++;
                        }// extra checking of boundry
                }// inner col loop
            }// outer row loop
        }// if boundry

        // convert LBP value to binary
        LBPValue = LBP[0] * (128) + LBP[1] * (64) + LBP[2] * (32)
            + LBP[3] * (16) + LBP[4] * (8) + LBP[5] * (4) + LBP[6] * (2)
            + LBP[7] * (1);

        // store the decimal LBP value in output array
        out[row * w + col] = LBPValue;
        atomicAdd(&histo[LBPValue], 1); // atomic operation to update histogram
}
```

3.2.2 Output images & Execution time

The results are for the size from 128 to 1024 (left to right):

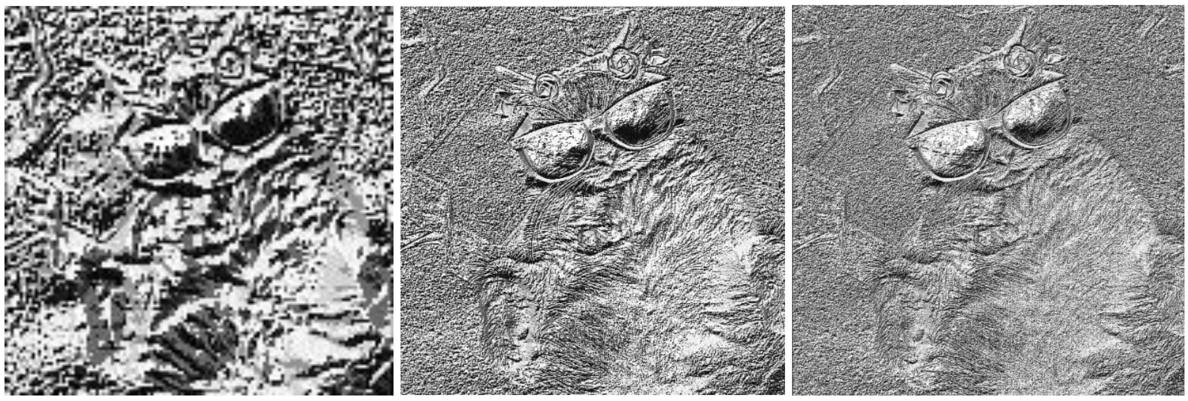

Figure 7 - LBP **GPU** output images

Table 2 - Execution time of the different inputs (GPU)

IMG SIZE	EXECUTION TIME (ms)
128X128	0.169
512X512	0.296
1024X1024	0.648

3.2.3 Histogram

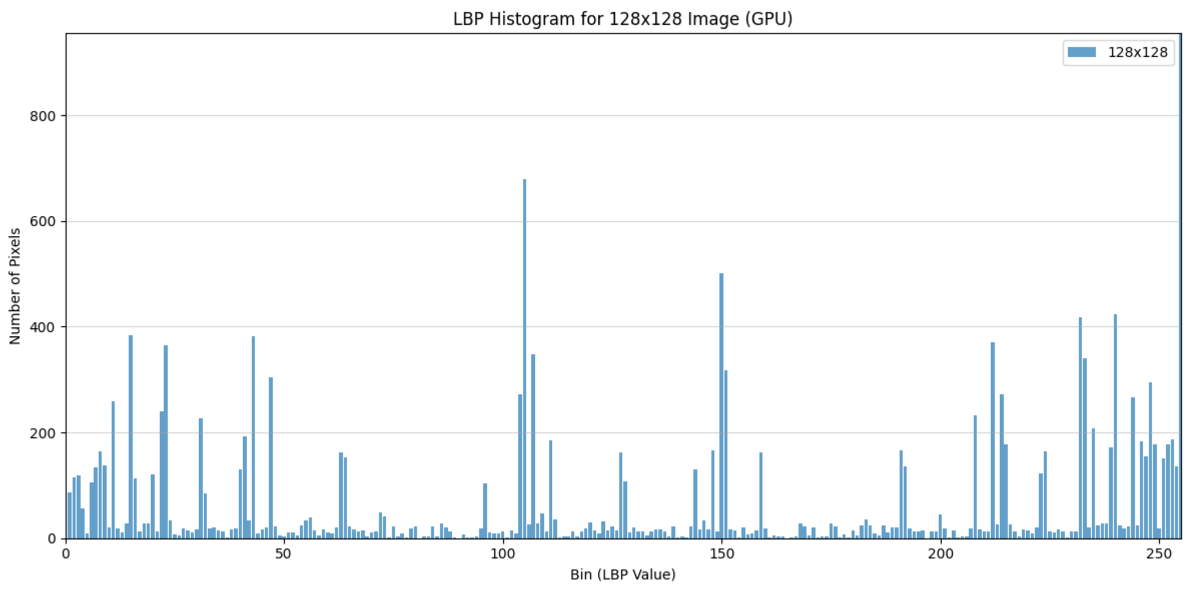

Figure 8 - GPU histogram 128x128

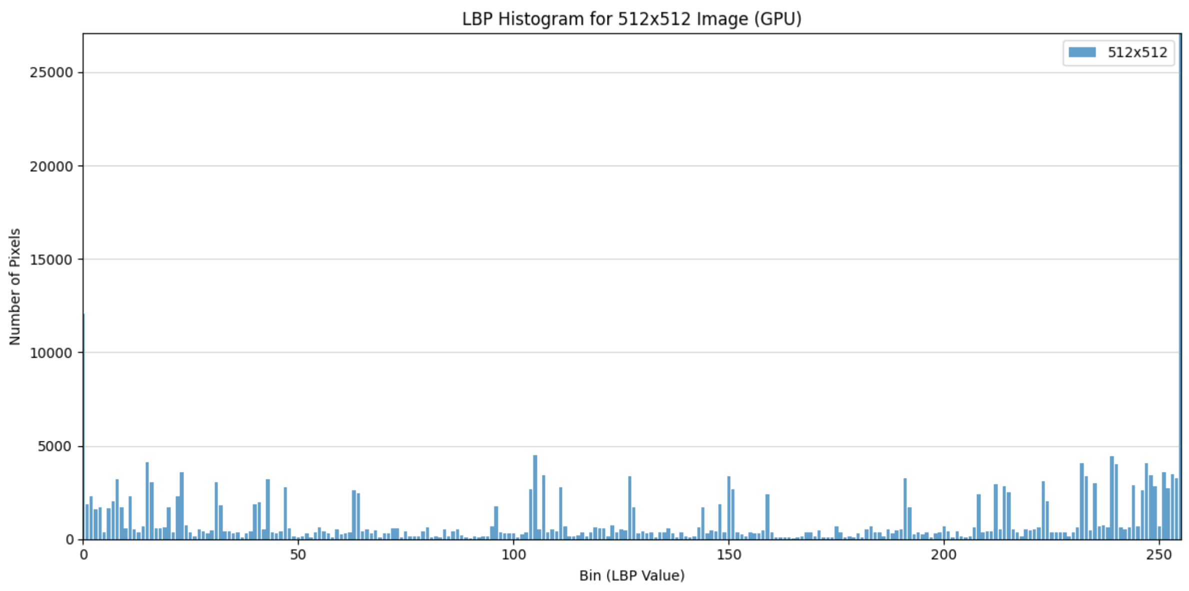

Figure 9 - GPU histogram 512x512

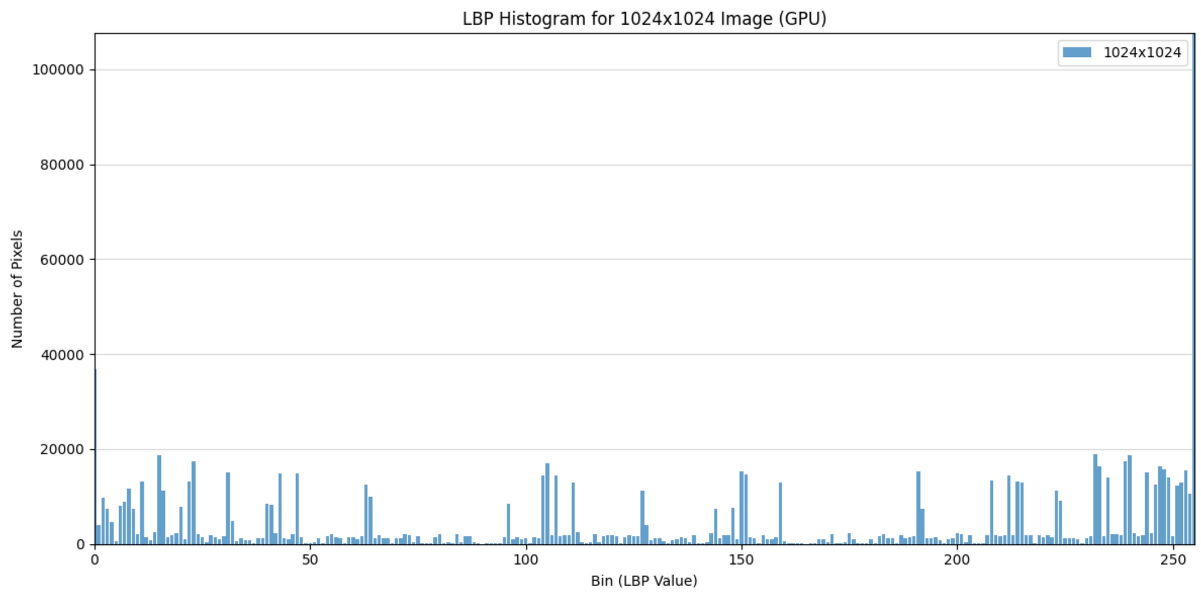

Figure 10 - GPU histogram 1024x1024

3.3 Optimized GPU Algorithm Results

3.3.1 CUDA kernel code

```
// GPU Kernel for LBP transformation with shared memory
__global__ void OptGPU_LBP(unsigned char* d_inArr, unsigned char* d_outArr, int* d_histo, int rows, int cols) {
    int bi = threadIdx.y;
    int bj = threadIdx.x;
    int i = blockIdx.y * blockDim.y + bi;  // Output image row
    int j = blockIdx.x * blockDim.x + bj;  // Output image col
    int padded_cols = cols + 2;  // Columns including padding, acts as new width

    __shared__ unsigned char shared_inArr[TILE_WIDTH * TILE_WIDTH];
// Shared memory for image block
    __shared__ int shared_histo[256];  // Shared histogram

    int tid = bi * BLOCK_WIDTH + bj;

    // Initialize shared histogram bins
    if (tid < 256)
        shared_histo[tid] = 0;

    // Load main TILE_WIDTH x TILE_WIDTH portion
// the startIndex is used as the 'base address' such that each thread can calculate //its own pixel in the shared memory
    int startIndex = (blockIdx.y * blockDim.y) * padded_cols + blockIdx.x * blockDim.x; // convert 2D to 1D
    int row = tid / TILE_WIDTH;
    int col = tid % TILE_WIDTH;
    int imgLocation = startIndex + (row * padded_cols) + col;

    // avoid reading out of bounds
    if (imgLocation < (rows + 2) * (cols + 2))
        shared_inArr[tid] = d_inArr[imgLocation];
    else
        shared_inArr[tid] = 0;

    // Load border pixels into shared memory
    if (tid < NUM_BORDER_PIXELS) {
        int border = tid + (BLOCK_WIDTH * BLOCK_WIDTH);
        // convert the border index to 2D (row/col) tjen scale it to the padded image
        row = border / TILE_WIDTH;
        col = border % TILE_WIDTH;
        imgLocation = startIndex + (row * padded_cols) + col;

        // check out of bounds
        if (imgLocation < (rows + 2) * (cols + 2))
            shared_inArr[border] = d_inArr[imgLocation];
        else
            shared_inArr[border] = 0;
    }

    __syncthreads();

    // Compute LBP value if within img bounds
    if (i < rows && j < cols) {
        int oldVal = shared_inArr[(bi + 1) * TILE_WIDTH + (bj + 1)];
        int newVal = 0;

        // Loop through 3x3 neighborhood
        for (int u = 0; u < 3; u++) {
            for (int v = 0; v < 3; v++) {
                // Compare pixel value with center
                if (shared_inArr[(bi + u) * TILE_WIDTH + (bj + v)] >= oldVal) {
```

```
                newVal += weights[u][v];  // Add corresponding weight
            }
        }
    }

    // Write result pixel to output
    d_outArr[i * cols + j] = newVal;

    // Use atomicadd to make sure no RACE conditions happen!
    atomicAdd(&shared_histo[newVal], 1);
  }

  __syncthreads();

  // Write shared histogram to global histogram
  if (tid < 256)
    atomicAdd(&d_histo[tid], shared_histo[tid]);
}
```

3.3.2 LBP Results & Execution time

The results are for the size from 128 to 1024 (left to right):

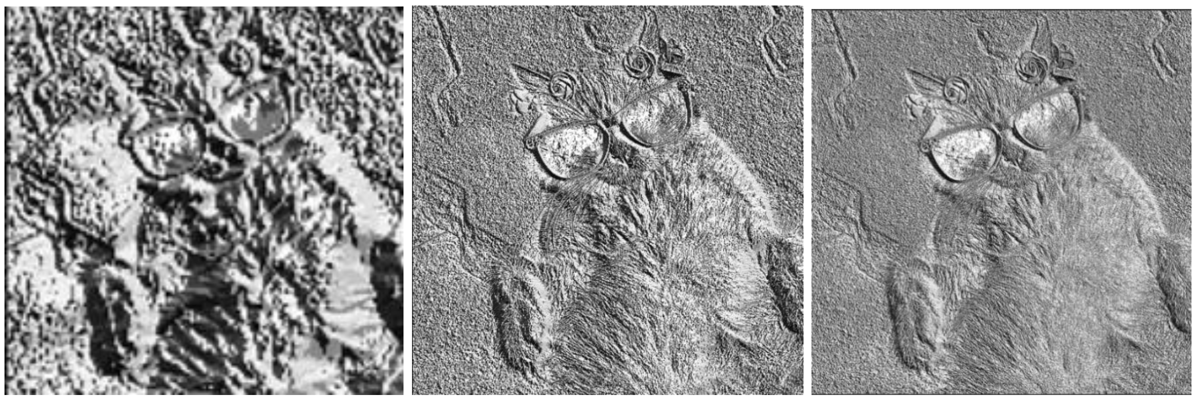

Figure 11 - LBP opt. GPU output images

Table 3 - Execution time of the different inputs (Opt. GPU)

IMG SIZE	EXECUTION TIME (ms)
128X128	0.097
512X512	0.18
1024X1024	0.357

3.3.3 Histogram

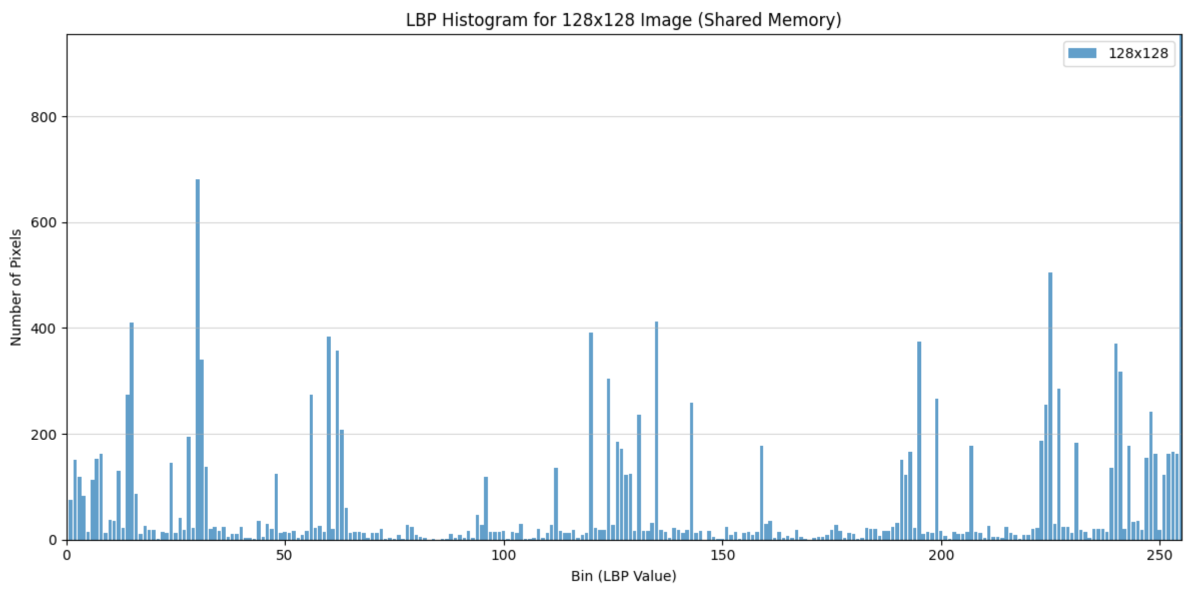

Figure 12 - Opt. GPU histogram 128x128

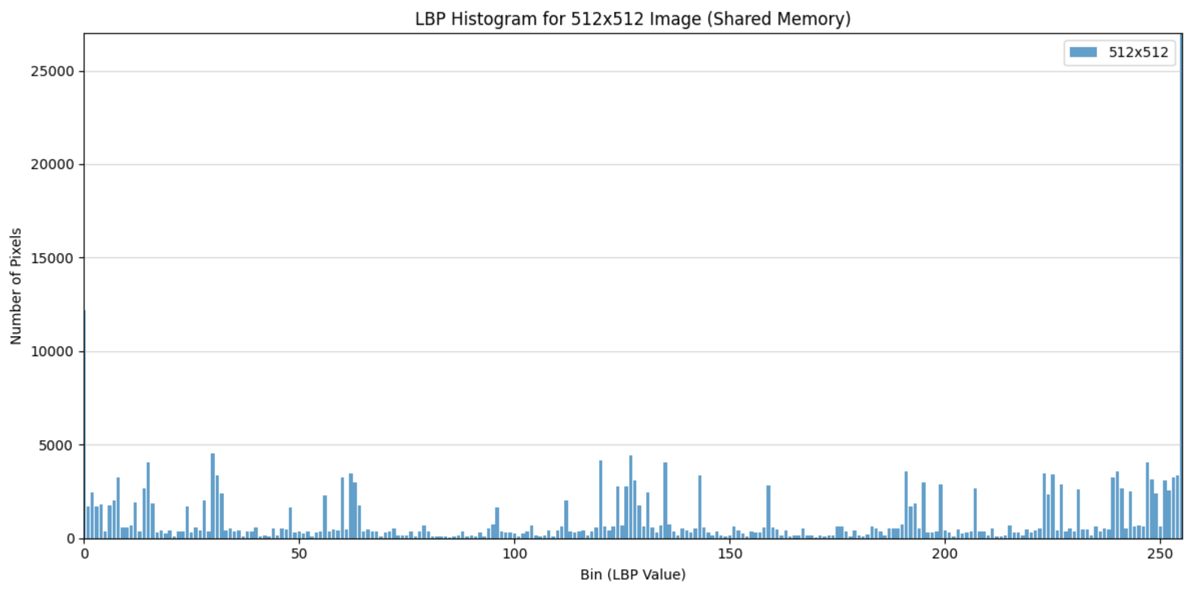

Figure 13 - Opt. GPU histogram 512x512

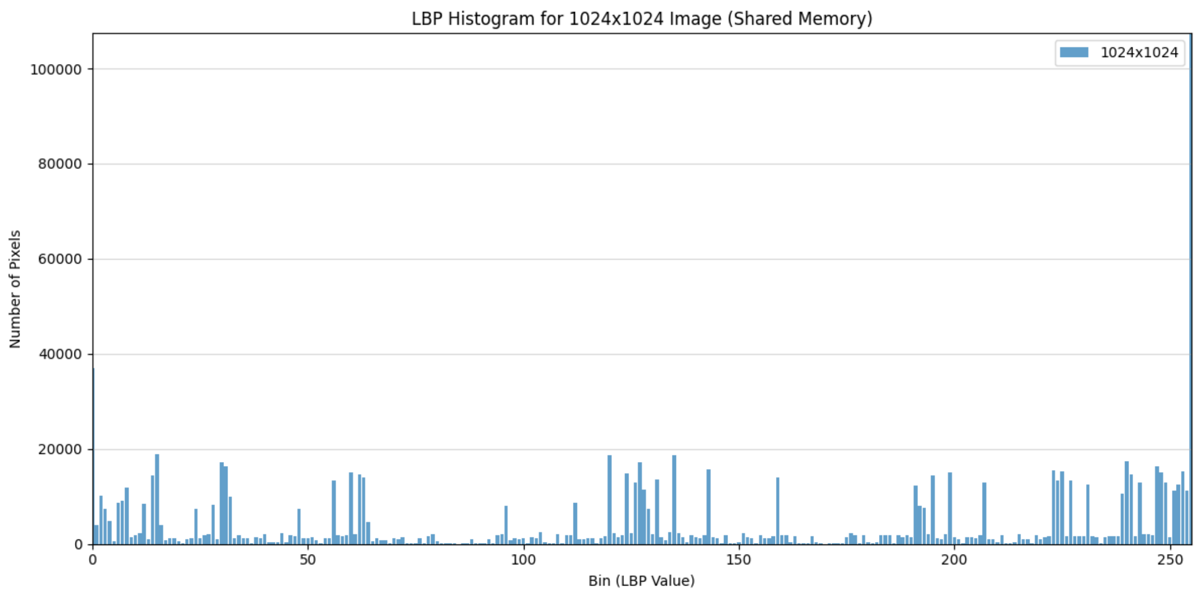

Figure 14 - Opt. GPU histogram 1024x1024

3.4 Histograms Analysis

The histogram analysis provides insight into the texture distribution of each image after applying the LBP transformation. It visualizes how frequently each LBP value (ranging from 0 to 255) occurs, showcasing repeated patterns in the image. The smaller the image the noisier the histogram is, that's because the pixels are limited and there isn't that much texture variation. As the image size increases, the histograms become smoother and more stable, with more consistent distributions. Additionally, across all algorithms (CPU, GPU, optimized GPU), the histograms show similar shapes for the same image size, indicating that the transformation is applied correctly and consistently regardless of implementation.

3.5 Speed Up Analysis

From the previous tables the following speedup can be calculated:

3.5.1 CPU Vs. GPU

$$Speedup\ Percentage = (\frac{CPU\ Time - GPU\ Time}{CPU\ Time}) \times 100$$

IMG SIZE	SPEED UP (CPU / GPU)
128X128	~0 (No speedup)
512X512	95.07%
1024X1024	97.05%

3.5.2 GPU Vs. Optimized GPU

$$Speedup\ Percentage = (\frac{GPU\ Time - Opt.GPU\ Time}{GPU\ Time}) \times 100$$

IMG SIZE	SPEED UP (GPU / OPT. GPU)
128X128	42.6%
512X512	39.2%
1024X1024	44.9%

3.5.3 Performance Analysis

The analysis of the execution times across CPU, GPU, and optimized GPU implementations reveals a clear trend in performance improvement with each step of optimization. The first table comparing CPU and GPU shows that while the GPU provides little to no benefit for very small images (128×128), it significantly outperforms the CPU as image size increases, achieving a 95.07% speedup for 512×512 images and a 97.05% speedup for 1024×1024 images. This shows that for larger images it’s better to use GPU.

implementation with the optimized version using shared memory and tiling highlights further performance gains. The optimized GPU achieves around 40–45% faster execution across all image sizes. This showcases that even though implementing the optimized algorithm is more difficult it’s worth it for a better performance with large scale images.

4. Conclusion

To conclude, the GPU is more complicated to implement than the CPU, but the efficiency and speedup of the GPU is worth the effort. For large data and faster results, the GPU is better because it is quicker in time. When comparing the histograms, we see the results of the LBP values that occurred the most. The optimized tiling GPU is the most complicated version, but it achieves the best performance in terms of execution time. Additionally, the image size significantly affects the clarity of the LBP results: smaller images tend to appear sharper and noisier due to fewer pixels representing fine details, while larger images produce clearer and more consistent LBP values and histograms across all algorithms.

References

[1] NVIDIA. (n.d.). *Using shared memory in CUDA C/C++*. NVIDIA Developer Blog. Retrieved May 4, 2025, from https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/

[2] NVIDIA. (n.d.). *CUDA C++ programming guide*. NVIDIA Developer Documentation. Retrieved May 4, 2025, from https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[3] Oak Ridge Leadership Computing Facility. (2019). *CUDA shared memory*. Oak Ridge National Laboratory. Retrieved May 4, 2025, from https://www.olcf.ornl.gov/wp-content/uploads/2019/12/02-CUDA-Shared-Memory.pdf

[4] Mickey's Tech Talk. (2021, May 20). *CUDA tutorial 3: Shared memory* [Video]. YouTube. https://www.youtube.com/watch?v=upGoZ00MlfI

[5] Pezzulla, S. (n.d.). *sim-pez (Simone Pezzulla)* [GitHub profile]. GitHub. Retrieved May 5, 2025, from https://github.com/sim-pez

[6] Stack Overflow. (2011). *When is CUDA's shared memory useful?* Retrieved May 8, 2025, from https://stackoverflow.com/questions/8011376/when-is-cudas-shared-memory-useful

[7] Kirk, D. B., & Hwu, W.-M. W. (2017). *Programming massively parallel processors: A hands-on approach* (3rd ed.). Morgan Kaufmann. from https://moodle.ku.edu.kw/course/view.php?id=94078#section-4

Appendices

Appendix A: Source Code

The source code can be found in this repository, please read the **README.md** file and follow the instructions in it such that you can run the code.

https://github.com/ProjectCodeKw/Computer-Arch.-Project